UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

A symbolic cost model for asynchronous parallel programs with structured dependences

Emmanuel Melin LIFO
Bruno Raffin ID-IMAG
Xavier Rebeuf LORIA
Bernard Virot Université d'Orléans

Rapport N^o **RR2002-02**

Abstract

We propose to associate a symbolic cost to a asynchronous parallel program. In contrast to classical approaches, we take into account all the dynamic asynchronism related to the different initial states that generate dynamic independences. We introduce an asynchronous intermediate model called $\mathcal{SCP}$. It only imposes static independences allowing a sequential lecture of programs. To compute the symbolic cost, we condition the existence of a dependence by annotating it with a predicate on program variables. To evaluate the cost in the initial environment, predicates are transformed by a backward method based on the model Weakest Liberal Preconditions Calculus. The resulting symbolic cost is parameterized by the initial environment and integrates dynamic dependences.
Keywords: Symbolic Cost Model; Weakest liberal preconditions; dynamic independences; Parallel Programming Languages.

Titre Français

Un modèle de coût symbolique pour les programmes parallèles asynchrones avec dépendances structurées

Résumé

Nous proposons d'associer un coût symbolique à un programme parallèle asynchrone. A contrario des approches classiques, nous prenons en compte tout l'asynchronisme dynamique relatif aux différents environnements initiaux possibles qui sont susceptibles de générer des indépendances dynamiques. Nous introduisons le modèle intermédiaire asynchrone $\mathcal{SCP}$. Il impose uniquement des indépendances statiques offrant ainsi une lecture séquentielle du programme. Afin de calculer le coût symbolique, nous conditionnons l'existence d'une dépendance en l'annotant par un prédicat sur les variables du programme. Pour évaluer le coût dans l'environnement initial, les prédicats sont transformés par une méthode "backward" basée sur le calcul des plus faibles pré-conditions libérales. Le coût résultant est paramétré par l'environnement initial et intègre les dépendances dynamiques.
Mots-Clefs: Modèle de coût symbolique; calcul des plus petites pré-conditions libérales; Indépendances dynamiques; Langages de Programmation Parallèle.

1 Introduction

Often, the goal of a parallelization is to minimize the execution time first. Models that evaluates a priori the cost of a given execution are essential in this context. Classical approaches like LogP [3] only associate a cost to a given execution based on an identification of the synchronizations (read/write dependences). To associate a cost to a program, we have to consider all possible dependences. Without restriction on the expression of the parallelism, an exact computation generally leads to a combinatorial explosion. To solve this problem, approaches like Athapascan [2] or BSP [15] generally restrict the parallelism by imposing static dependences between instructions. This allows to evaluate statically a coherent superset of the dependences of all the executions of a program. Nevertheless, the cost evaluation is not accurate, as these models do not take into account all the dynamic asynchronism related to the different initial states that generate dynamic independences (i.e. absence of dependence).

In previous papers [12] we proposed an approach of dependences structuring in the framework of distributed memory architectures. In [10] we showed that it is possible to define a complexity function in this framework. It yields a symbolic date for each communication event. These dates can be ordered to compute upper bounds for the network load. We proposed a cost model able to handle static asynchronism in message passing and communication/computation overlap.

Our purpose is to describe a cost model refining these classical approaches. We introduce an intermediate model called $\mathcal{SCP}$. Like BSP, the $\mathcal{SCP}$ programming model allows a sequential lecture of programs. Like Athapascan, the execution model is asynchronous. But in contrast to Athapascan, $\mathcal{SCP}$ only imposes static independences (i.e. we forbid the corresponding dependences). We introduce a logical ordering on instructions relying on the program structure. Because all processors, called indices, run the same program, the logical ordering is a shared information. We use it to allow dynamic resolution of data accesses and synchronizations. This leads to deterministic, efficient and deadlock-free asynchronous executions [9]. We show that it can be adapted to the classical framework of PRAM shared memory architecture.

We show it is possible to represent symbolically the set of effective dependences for each execution, and thus to obtain an exact evaluation of the cost while avoiding combinatorial explosion. We condition the existence of a dependence by annotating it with a predicate on program variables. The program cost is then expressed in a symbolic way with regard to these variables. To evaluate the cost in the initial environment, predicates are transformed by a backward method based on the model Weakest Liberal Preconditions Calculus of Dijkstra. The resulting symbolic cost is parameterized by the initial environment and integrates dynamic dependences.

2 The $\mathcal{SCP}$ model

We first present the $\mathcal{SCP}$ language, the logical ordering and data dependences management. Next we describe the $\mathcal{SCP}$ denotational semantics.

2.1 The language

The $\mathcal{SCP}$ language is based on the SPMD paradigm. Indices execute instructions in an asynchronous way. An index is *active* for an instance of instruction if it runs it. The number p of indices is a constant.

The $\mathcal{SCP}$ language has scalar variables. All $\mathcal{SCP}$ programs respect the *owner compute rule*: an active index executes an assignment only if it owns the left part variable.

Assignment $X := Exp$. The index owning X evaluates the expression Exp and then assigns the obtained value to X.

Note that the name space is global and the evaluation of an expression may introduce implicit communications. Moreover the left part variable of an assignment may takes the shape of an array element: $X[Exp_2] := Exp$. In this case all active indices evaluates the expression Exp_2 to know if they own $X[Exp_2]$ and then if they perform the assignment.

We introduce the vectorial constant $This$. An index evaluating $This$ obtains its own index value. Let us consider that the vectorial constant $This$ appears in a given expression Exp. When two indices evaluates Exp results may be differents. We denotes by $Exp|_u$ the value obtained when Exp is evaluated by the index u. For example the expression $This$ evaluated by the index u gives $This|_u = u$.

Remark.

- The model can be extended with a structure similar to a BSP superstep [15] or a task in Athapascan [2]. A sequence of instructions of a sequential host language may replace each assignment in the $\mathcal{SCP}$ language. In this case, $\mathcal{SCP}$ can be seen as a coordination SPMD language of sequential programs [9].

Sequence: $S;T$. When an active index ends the execution of S, it starts the execution of T.

To introduce task-parallelism, a conditional structure divides the abstract machine into two independents sub-machines. A stack mechanism keeps all values assigned to each variable. This suppresses all dependences between sub-machines.

Concurrent conditioning: where B do S elsewhere T end. If an active index u evaluates the condition B to $true$, then it executes the program S, otherwise it executes T.

A single branch conditioning construct where B do S end may be defined as where B do S elsewhere step $\{\}$ end.

Loop: loopwhere B do S end. Iteration is expressed by classical loop unfolding. An active index repeatedly executes S while it evaluates the expression B to $true$.

We introduce the structure forwhere $I[This] := a$ to b do S end where a and b are constants and the array I is distributed onto indices such as $I[this]$ is locally evaluated for each index. It is equivalent to a loop loopwhere as we illustrate in the following translation.

forwhere $I[This] := a$ to b do
  S
end
	$I := a$
	loopwhere $I[This] \leq b$ do
	  S;
	  $I[This] := I[This] + 1$;
	end

2.2 Data dependences management

To execute asynchronously a parallel program, we have to manage different kinds of dependences between indices :

- Sequence dependences. Each index has to execute all its instructions in the natural sequential order. Sequence dependences must be respected to preserve the calculus semantics.
- Flow dependences. A index u has to wait another index v if v has to write a variable X before u reads it. Flow dependences must be respected to preserve the calculus semantics.
- Anti-dependences. An index u has to wait another index v if v has to read a scalar X before u writes it.

We first define a logical ordering between the instances of the $\mathcal{SCP}$ program instructions. This ordering models the static independences expressed by the $\mathcal{SCP}$ program structure. We next show how the $\mathcal{SCP}$ execution model uses this ordering to dynamically set-up the synchronizations required to respect the sequence and flow dependences, the anti-dependences being suppressed by a classical single assignment mechanism [9].

2.2.1 Instruction structural ordering

We define a strict ordering on instruction instances [7, 12] based on the program structure. We denote by $Inst(S)$ the set of instruction instances of S. We define the strict ordering $<_S$ from $Inst(S) * Inst(S)$ to booleans by induction onto the program structure.

Instructions. The program S is reduced to one single instruction. In this case, the set of instruction instances $Inst(S)$ is the singleton $\{i\}$. As we want a strict ordering, we define the order onto this singleton as:

$$(i \ <_S \ i) \ = \ False$$

Observe that incomparability of an instance with itself modelizes the independence between two executions of this instance by two different indices.

Sequence. The program S is a sequence $R; T$. We introduce an ordering between each instance $i_1 \in Inst(R)$ and $i_2 \in Inst(T)$.

$$\begin{aligned}(i_1 \ <_S \ i_2) \ = \ & (i_1 \in Inst(R) \wedge i_2 \in Inst(R) \wedge i_1 <_R i_2) \\ & \vee (i_1 \in Inst(T) \wedge i_2 \in Inst(T) \ \wedge i_1 <_T i_2) \\ & \vee (i_1 \in Inst(R) \wedge i_2 \in Inst(T))\end{aligned}$$

The ordering $<_S$ onto $Inst(R)$ (resp. $Inst(T)$) extends the orderings $<_R$ and $<_T$.

Concurrent conditioning: The program S is a conditioning where B do R elsewhere T end. We forbid dependencies between an instruction i_1 from R and an instruction i_2 from T. Notice that we consider where B do and end as two different instructions.

$$\begin{array}{rcl}(i_1 \ <_S \ i_2) & = & (i_1 \in Inst(R) \wedge i_2 \in Inst(R) \wedge i_1 <_R i_2) \\ & & \vee\ (i_1 \in Inst(T) \wedge i_2 \in Inst(T) \wedge i_1 <_T i_2) \\ & & \vee\ (i_1 = \text{where } B \text{ do } \wedge i_2 \in Inst(R) \cup Inst(T)) \\ & & \vee\ (i_1 \in Inst(R) \cup Inst(T) \wedge i_2 = \text{end})\end{array}$$

The order $<_S$ onto $Inst(R)$ (resp. $Inst(T)$) extends the order $<_R$ (resp. $<_T$). Observe that the order $<_S$ is partial since an instruction instance from R is not comparable with an instruction instance from T. This modelizes the independence of the two branches of a same conditioning.

Loop. The instruction order for a loop can be deduced from preceding rules by classic loop unfolding.

For all programs S, we define the first instance start such as for each instance $i \in Inst(S)$, we have start $<_S \ i$.

Remark. Note that two distinct instruction instances are comparable if and only if they can be executed by the same index.

Thanks to the structural ordering, we can define a dynamic data access rule. We introduce a *Waiting condition* based on the distribution of data and on the program structure [7, 12]. Data are distributed according to the owner function $\mathcal{O}w$. We denote by $\mathcal{O}w(X)$ the index owning the variable X.

Remark. In $\mathcal{O}w(X)$, X denotes the variable name but not its value.

Waiting condition. *Let S be a $\mathcal{SCP}$ program and u an index that executes an instruction instance $inst_u \in Inst(S)$. If u* **reads** *a variable X, then it has to wait that the index $v = \mathcal{O}w(X)$ had executed all instances $inst_v \in Inst(S)$ such as $inst_v <_S inst_u$.*

2.2.2 Memory accesses

In order to suppress anti-dependences, we introduce a stack memory mechanism. Each variable X can be seen as a set of pairs (V, i) composed of a value V and an instruction instance i denoted $Stack(X)$. An initial value is given to each variable labeled with the instruction instance start. Assignments do not erase previous values.

Write accesses $X := Exp$**.** The index $\mathcal{O}w(X)$ pushes on $Stack(X)$ the value V corresponding to the evaluation of $Exp|_{\mathcal{O}w(X)}$ labeled by the current assignment instance.

Read accesses. As soon as the waiting condition is satisfied, the index referring to the variable X reads the value labeled by the greatest instruction instance preceding (strictly) the current instance on $Stack(X)$.

As proved in [9] the $\mathcal{SCP}$ model is deterministic and deadlock free.

The dependences can be represented by a classic dependence graph. Each node ($inst$,u) corresponds to an instruction $inst$ executed by an index u. Each arrow from the node ($source$,u) to the node ($target$,v) symbolizes a dependence between the execution of $source$ by u and the execution of $target$ by v.

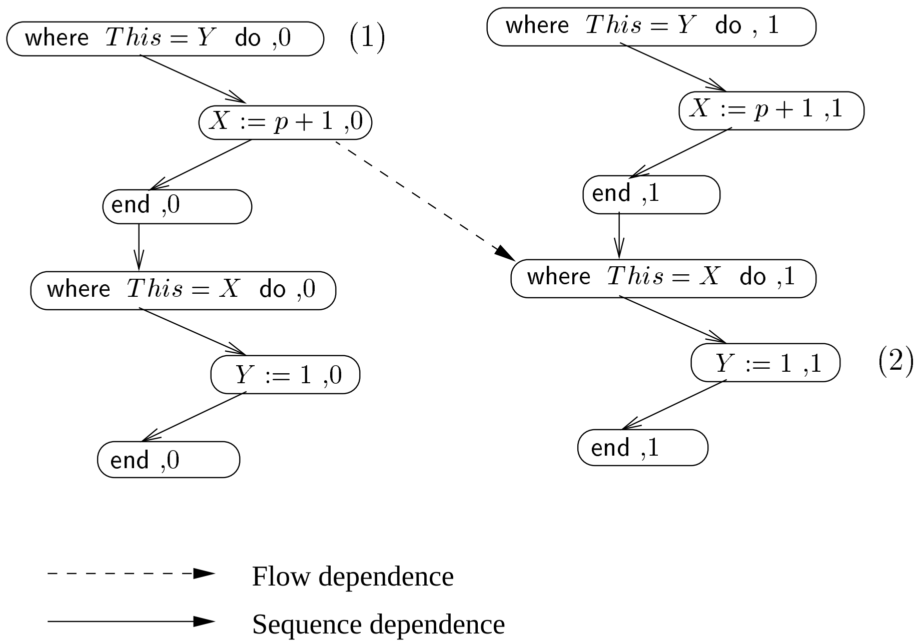
Example 1 *(Running example)*

Consider the following S program, where p denotes the number of indices.

```
where This = Y  do
   X := p + 1
end;
where This = X  do
   Y := 1
end
```

Only the index Y runs into the first conditioning. If it owns the X variable, then it assigns $p+1$ to X. Before running the second conditioning, all indices have to respect the flow dependence between the assignment of X and the reading of X in the second where*. Hence they all wait for the owner of X. Then, the index corresponding to the new value of X runs into the second conditioning.*

Let us now consider the execution where $\mathcal{O}w(Y) = 1$ and $\mathcal{O}w(X) = 0$. We obtain the following dependence graph.



Remark. *The anti-dependence from the instruction* (1) *to the instruction* (2) *is suppressed by the memory management : the initial value of Y is not erased by the assignment $Y := 1$.*

2.3 Denotational semantics

The cost model is based on the denotational semantics of $\mathcal{SCP}$, presented in details in [9]. For sake of completeness we briefly describe it now.

A state, denoted σ, is a function returning for each variable its value in memory. For sake of simplicity, we introduce a vectorial variable $\#$, symbolizing the context in the current state [6]. The evaluation of $\#|_u$ in the current state is $True$ if and only if the index u is active. We denote $MemEnv$ the set of all states. We modelize never ending program executions with the special value $\perp$.

The denotational semantics links each program S with the function it computes, noted $[\![S]\!]$. The $[\![S]\!]$ function is defined by induction on the program structure. From an element of $MemEnv \cup \{\perp\}$, it returns one element of $MemEnv \cup \{\perp\}$. For all programs S, $[\![S]\!](\perp) = \perp$.

- **Skip.** This instruction does nothing and terminates.

$$[\![\mathsf{skip}]\!]\sigma = \sigma$$

- **Assignment.** The instruction $X[Exp1] := Exp2$ assigns $\sigma(Exp2)$ to the variable $X[\sigma(Exp1)]$ only if its owner is active.

$$[\![X[Exp1] := Exp2]\!](\sigma) = \sigma'$$

$$\sigma'(Y) = \begin{cases} \sigma(Exp2|_{\mathcal{O}w(Y)}) \text{ si } Y = X[\sigma(Exp1|_{\mathcal{O}w(Y)})] \wedge \sigma(\#|_{\mathcal{O}w(Y)}) \\ \sigma(Y) \text{ sinon.} \end{cases}$$

- **Sequence.** $S; T$. All active indices run S and then T.

$$[\![S; T]\!](\sigma) = [\![T]\!]([\![S]\!](\sigma)).$$

- **Concurrent conditioning.** $\mathsf{where}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{elsewhere}\ \ T\ \ \mathsf{end}$. The final state is a merging of branch final states each computed thru the initial state.

$$[\![\mathsf{where}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{elsewhere}\ \ T\ \ \mathsf{end}]\!](\sigma) = (\sigma'),$$

with

$$\sigma_S(X) = \begin{cases} \sigma(\#) \wedge \sigma(B) \text{ if } X = \#, \\ \sigma(X) \text{ else.} \end{cases} \quad \Bigg| \quad \sigma_T(X) = \begin{cases} \sigma(\#) \wedge \neg\sigma(B) \text{ if } X = \#, \\ \sigma(X) \text{ else.} \end{cases} \quad ,$$

and

$$\sigma'(X) = \begin{cases} \sigma(\#) \text{ if } X = \# \\ [\![S]\!](\sigma_S)(X) \text{ if } \sigma(B|_{\mathcal{O}w(X)}), \\ [\![T]\!](\sigma_T)(X) \text{ else.} \end{cases}$$

- **Loop.** $\mathsf{loopwhere}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end}$. When loop ends, denotational semantics is based on the least fixed point [14], denoted $fix(F)$, of the function F defined by:

$$F \ = \ \lambda f\ \lambda\sigma\ if\ (\exists u\ such\ as\ \sigma(\#|_u))\ then\ f([\![S]\!](\sigma_f))\ else\ \sigma$$

with

$$\sigma_f(X) = \begin{cases} \sigma(\#) \wedge \sigma(B) \text{ if } X = \#, \\ \sigma(X) \text{ else.} \end{cases}$$

We deduce the loop semantics

$$[\![\mathsf{loopwhere}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end}]\!](\sigma) = \sigma'$$

with

$$\sigma'(X) = \begin{cases} \sigma(\#) \text{ if } X = \#, \\ Fix(F)(\sigma)(X) \text{ else} \end{cases}$$

If the loop does not end, we take the common practice

$$[\![\mathsf{loopwhere}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end}]\!](\sigma) = \perp.$$

3 A symbolic cost model

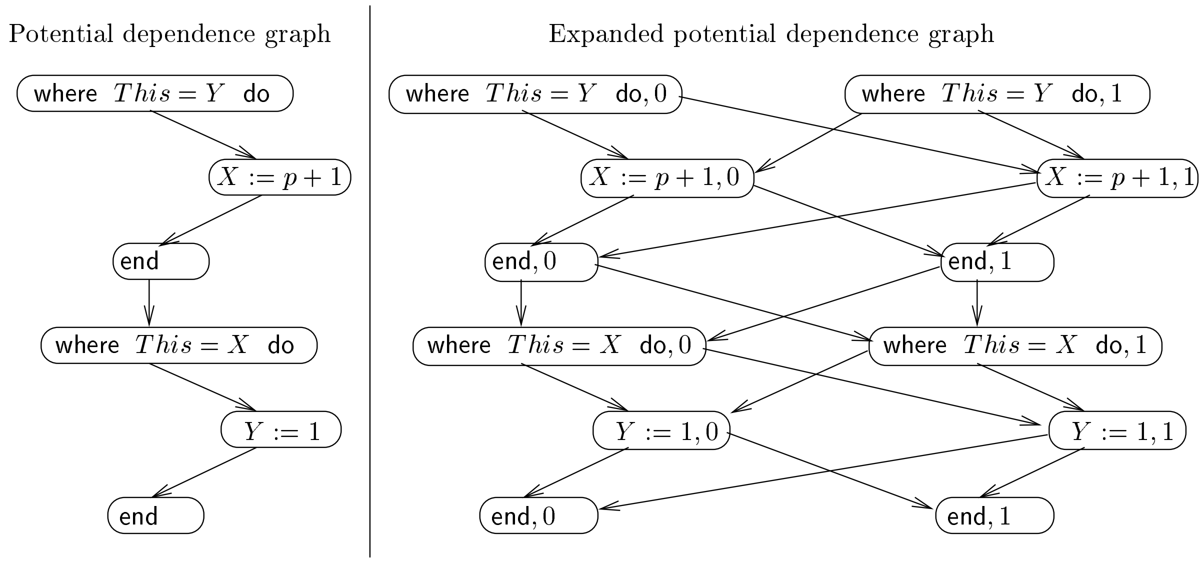
We define a cost model for a $\mathcal{SCP}$ program. It has to take into account the static independences defined by the program as well as the dynamic independences arising with regard to the initial data. Static independences are modelized by a potential dependence graph. Each potential dependence is conditioned by a predicate that represents dynamics independences.

3.1 The potential dependence graph

We model static independences by a graph. They are only defined with regard to the structure of the program. We first consider the complementary set of static independences that we call the set of potential dependences. The existence of a potential dependence from the node *source* to the node *target* means that a dependence may occur from the execution of the instruction *source* by an index to the execution of *target* by an other one. We construct the graph of potential dependences where each node corresponds to an instruction *inst*. Each arrow from the node *source* to the node *target* symbolizes a potential dependence. The graph is built unfolding the loops.

Because dynamics independences are index dependent we introduce an expanded version of the graph where each node *Inst* in $(Inst, u)$ for each index u is duplicated. By construction, for a given program, the expanded potential dependence graph covers all the possible dependence graph.

Example 2 *To illustrate the graph construction, we reuse the example 1.*



The expanded potential dependence graph covers the dependence graph presented page 5.

Because the potential dependence graph matches exactly the partial order on the instances of the program instructions, it is a Direct Acyclic Graph.

3.1.1 The cost of an execution

In order to compute the cost of an execution (i.e. for a given program S and a given initial state σ), as usual [3] we have to construct a classical execution graph describing actual dependences. In this section, we show that it can be deduced from the potential dependence graph.

First, we suppress from the graph each node $(inst, v)$ satisfying one of the two following conditions:

- The instruction *inst* is not meaningful, i.e. *inst* is only a syntactic marker (end).
- The index v is not active with regard to *inst* in the corresponding execution.

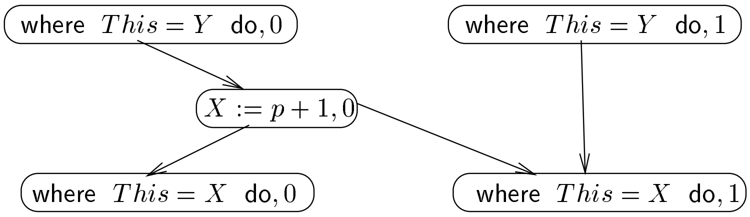
Because $inst$ is not executed by v, we can suppress all the dependences needed to execute $inst$, i.e. the arrows ending at this node. But, we have to preserve all the dependences needed to execute other instructions, i.e. the set S of the arrows starting from this node. Therefore, we link each arrows of S to each predecessor $(inst, u)$ where $u = v$.

Finally, we suppress each arrow from $(source, u)$ to $(target, v)$ corresponding to a flow dependence that does not occur in the execution (i.e. where $u \neq v$ and, where there is no variable of u accessed during the execution of $target$ by v).

In the resulting graph, each node $(inst, u)$ corresponds to an instruction $inst$ actually executed by an index u. Each arrow from the node $(source, u)$ to the node $(target, v)$ symbolizes a effective dependence occurring from the execution of $source$ by u to the execution of $target$ by v.

Thanks to this graph, we classically define a LogP [3] like cost, denoted by $Cost_e(S, \sigma)$, as the length of the longest path with regard to the nodes.

Example 3 *Let turn to the running example 1. We consider the execution corresponding to the initial state σ where $\sigma(X) = 1$ and $\sigma(Y) = 0$, and the distribution $\mathcal{O}w$ where $\mathcal{O}w(Y) = 1$ and $\mathcal{O}w(X) = 0$. We obtain the following dependence graph.*



From the execution graph presented in the example 3, we deduce $Cost_e(S, \sigma) = 3$.

3.2 Conditioning potential dependences

Our aim is to associate a cost to a program. In order to avoid combinatorial explosion, we cannot construct a graph for each execution of the program. We propose to use the potential dependence graph. However, it does not take into account dynamic independences. Therefore, we label each potential dependence from the instruction $source$ to the instruction $target$ by a Boolean expression $C(u, v)$ (where u and v are indices). The evaluation of the label $C(u, v)$ in the current state (i.e. before the execution of $target$ by v) conditions the existence of a dependence from the execution of $source$ by u to $target$ by v.

In order to construct the labels, we reuse the assertion language defined in [6]. This language manages scalar variables and parallel state variables. $X|_u$ denotes the component of parallel variable X located at the index u. The parallel Boolean variable $\#$ denotes the activity. The variable $\#|_u$ is evaluated to $True$ if and only if the index u is active.

We introduce a new conditional expression $B?S : T$ evaluated to S if the condition B is $True$, and to T otherwise. If there is no confusion, we denote by $B?S$ the expression $B?S : 0$.

The labels are usual first order formulae. They are recursively defined on Boolean expressions with logical and arithmetical operators. The basic expressions are variables and constants.

3.2.1 The label construction

In order to construct labels, we use the potential dependences graph and the denotational semantics. Let $C(u, v)$ be a label for a potential dependence from $Source$ to $Target$. Intuitively, this label is evaluated to $True$ if and only if there exists a dependence from the execution of $Source$ by u to the execution of $Target$ by v.

When the index v executes the instruction $Target$, it refers to a set of variables, denoted by $Used(Target, v)$. We first construct this set. We denote by $Var(Exp)$ the set of all the variables appearing in Exp. We distinguish three cases depending on the instruction $Target$.

- If $Target$ is a conditional structure where B do, then v has to evaluate the expression B.

$$Used(Target, v) = Var(B|_v)$$

- If $Target$ is a assignment $X[Exp1] := Exp2$, v has to evaluate the expression $Exp1$. Then, if evaluates the expression $Exp2$ only if it owns the variable $X[Exp1]$.

$$Used(Target, v) = Var(Exp1|_v) \cup \{X/X \in Var(Exp2|_v) \wedge v = \mathcal{O}w(X[Exp1|_v])\}$$

- Otherwise, the set is empty.

$$Used(Target, v) = \emptyset$$

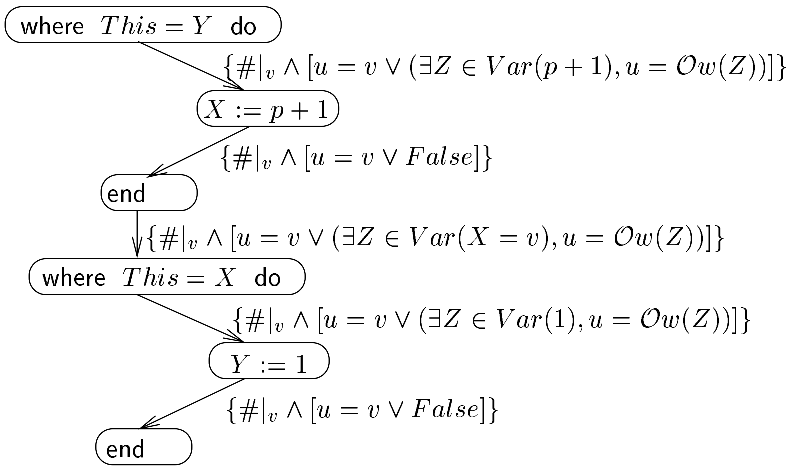
We turn now to the construction of the condition $C(u, v)$. First, a dependence occurs only if the index v is active $(\#|_u)$. Next, we distinguish the sequence dependence and the flow dependence.

- A sequence dependence occurs if and only if the target index is the source index $(u = v)$.
- A flow dependence occurs if and only if the index v has to wait for the index u, i.e. u owns a referred variable $(\exists X \in Used(Target, v),\ u = \mathcal{O}w(X))$.

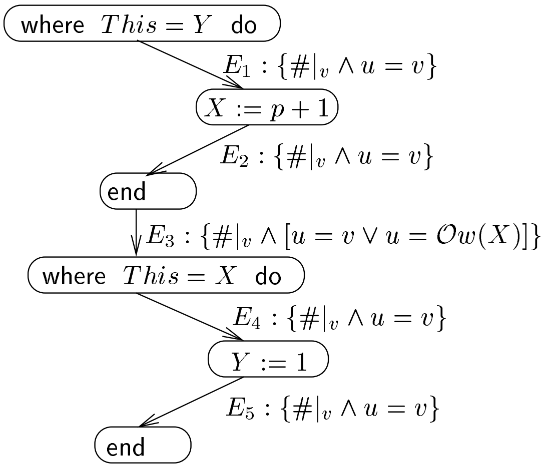
We obtain the following label.

$$C(u, v) = \#|_v \wedge [(u = v) \vee (\exists X \in Used(Target, v),\ u = \mathcal{O}w(X))],$$

Example 4 *Let reuse the running example 1. We obtain the following labeled graph.*



To simplify, some reduction rules can be applied on symbolic expressions [13]. For example, as $p + 1$ is a constant, there is no variable belonging to this expression. Therefore, the expression $u = v \vee (\exists Z \in Var(p + 1), u = \mathcal{O}w(Z))$ can be reduced in $u = v$. We obtain the following labeled graph where each label is called E_i for convenience.



Remark. If the distribution is not defined, the $\mathcal{O}w$ function is not evaluated and remains as a symbol in the labels.

3.2.2 Evaluating labels

Each label refers to the current state. Therefore, for a given initial state, we have to compute the corresponding current state thanks to the denotational semantics. If a label occurs in a branch where (resp. elsewhere) of a conditional structure, we have to build the current state in this structure. We propose to extend the denotational semantics in order to compute the resulting state inside a conditional structure. The following rule only modifies the activity.

$$[\![\text{where } B \text{ do}]\!](\sigma) = \sigma'$$

where

$$\sigma'(X) = \begin{cases} \sigma(\#) \wedge \sigma(B) \text{ if } X = \# \\ \sigma(X) \text{ otherwise.} \end{cases}$$

From this rule, we can deduce the following ones

$$\begin{array}{l} [\![\text{where } B \text{ do } S]\!](\sigma) = [\![S]\!]([\![\text{where } B \text{ do}]\!](\sigma)) \\ [\![\text{where } B \text{ do } S \text{ elsewhere}]\!](\sigma) = [\![\text{where } \neg B \text{ do}]\!](\sigma) \\ [\![\text{where } B \text{ do } S \text{ elsewhere } T]\!](\sigma) = [\![\text{where } \neg B \text{ do } T]\!](\sigma) \end{array}$$

Remark. Since there is no rule for end and elsewhere, the computation of the denotational semantics may not terminate. In order to ensure the termination, we have to apply the rules with the following priority order.

where B do S elsewhere T end
where B do S elsewhere T
where B do S end
where B do S
where B do

3.3 Transforming the labels regarding to the initial state

We want to associate a symbolic cost to the labeled graph corresponding to the length of the longest path. Each label refers to a different current state. Therefore, we have to transform them with regard to a single state: The initial state σ_0. We use a "backward" method based on the weakest liberal precondition [4].

We classically define the weakest liberal precondition of a program S with regard to a set of states R, denoted by $wlp(S, R)$, as the set of state σ such as, for all execution of S initiated in an state σ, if the execution ends, then the resulting state belongs to R.

Definition 1 *Let R be a set of states, and S be a $\mathcal{SCP}$ program. We define the weakest liberal precondition of S with regard to R as follows*

$$wlp(S, R) = \{\sigma \;/\; [\![S]\!](\sigma) \in R \cup \{\perp\}\}$$

A state σ satisfies a label P, denoted by $\sigma \models P$, if P is evaluated to True in the state σ. By convention, the indefinite state $\perp$ satisfies all the labels. If there is no confusion, we merge the predicate P and the set of states σ, such as $\sigma \models P$.

To evaluate a label Q in the initial state of a program S, we define the equivalent precondition P with regard to S as the label satisfying the following condition: the label Q is evaluated to true after the execution of S if and only if the precondition P is evaluated to True in the initial state.

Definition 2 *A precondition P is equivalent to Q with regard to S, denoted by $\{P\}S\{Q\}$, if the two following conditions are equivalent:*

- $\sigma \models P$,
- $\sigma' = [\![S]\!](\sigma) \models Q$.

The following theorem shows that the precondition P corresponds to the weakest liberal precondition of a program S with regard to Q ($wlp(S, Q)$).

Theorem 1 *Let S be a $\mathcal{SCP}$ program, P and Q two labels. We have $\{P\}S\{Q\}$ if and only if $P = wlp(S, Q)$.*

Proof 1 *The proof is deduced from the precondition definition. We have $\{P\}S\{Q\}$ if and only if for each state σ, $\sigma(P) = [\![S]\!](\sigma)(Q)$. P can be expressed as an set of states: $P = \{\sigma \;/\; [\![S]\!](\sigma)(Q)\}$. If we express Q as a set of states, we obtain $P = \{\sigma \;/\; [\![S]\!](\sigma) \in Q\}$. By definition, $\perp$ satisfies all the labels. It belongs to the set Q. Therefore, we have $P = \{\sigma \;/\; [\![S]\!](\sigma) \in Q \cup \perp\}$.*

Let us present the computing rules of the weakest precondition for each language structure.

The skip instruction. We deduce from a label P the equivalent label before this instruction. As it just does nothing but terminates, the predicate remains the same.

Proposition 1 *For a given label P, we have:*

$$\{P\}\ \mathsf{skip}\ \{P\}$$

Proof 2 *The proof relies on the denotational semantics of the instruction skip presented in 2.3. Let σ be a state such as $\sigma \in wlp(\mathsf{skip}, P)$. Therefore, $\sigma = [\![\mathsf{skip}]\!](\sigma) \models P$. Conversely, if $\sigma \models P$ then $\sigma \in wlp(\mathsf{skip}, P)$. This corresponds to the application of the theorem 1.*

The assignment $Y := Exp$. In order to express the weakest precondition of an assignment, we define a substitution operator for the variables. Let P be a label, X a variable, and E an expression. We denote by $P[E/X]$ the resulting expression of the substitution by E of all the instances of X appearing in P.

For the assignment rule, we use the substitution lemma [1]. The variable Y is modified only if the owner of Y is active. Therefore, we condition by $\#|_{\mathcal{O}w(Y)}$ the substitution of Y by P. We first suppose that Exp contains only scalar variables.

Proposition 2 *Let P be a label, we obtain:*

$$\{P[\#|_{\mathcal{O}w(Y)}?Exp : Y/Y]\}\ Y := Exp\ \{P\}$$

Proof 3 *The proof is similar to those presented for the languages $\mathcal{L}$ and $\mathcal{L}'$ in [8]. It relies on the denotational semantics presented in 2.3. Let $\sigma \in wlp(Y := Exp, P)$. Then $\sigma' = [\![Y := Exp]\!](\sigma) \models P$.*

By definition of the denotational semantics, we have

$$\sigma'(Z) = \begin{cases} \sigma(Exp) \text{ if } Z = Y \wedge \sigma(\#|_{\mathcal{O}w(Y)}), \\ \sigma(Z) \text{ otherwise.} \end{cases}$$

The substitution lemma [1] is applicable to our framework [5, 6]. We obtain $\sigma \models P[\#|_{\mathcal{O}w(Y)}?Exp : Y/Y]$. Conversely, let σ be a state such as $\sigma \models P[\#|_{\mathcal{O}w(Y)}?Exp : Y/Y]$. Thanks to the substitution lemma, we have $\sigma' = \sigma(P[\#|_{\mathcal{O}w(Y)}?Exp : Y/Y]) \models P$. By using the denotational semantics, we obtain: $[\![Y := Exp]\!](\sigma) = \sigma' \models P$. The proposition is verified.

Example 5 *Let S be an $\mathcal{SCP}$ program reduced to one assignment $X := 3$. We calculate the weakest precondition corresponding to the label $\{X = Y\}$.*

$$\begin{array}{l} \{[\#|_{\mathcal{O}w(X)}?3 : X] = Y\} \\ X := 3 \\ \{X = Y\} \end{array}$$

The assignment rules is applied to an array element $X[Exp1] := Exp2$. For each active index u owning $X[Exp1]$, we substitute in P the instances of $X[Exp1]$ by $Exp2$.

$$\begin{array}{c} \{P[\#|_u \wedge u = \mathcal{O}w(X[Exp1])?Exp2 : X[Exp1]/X[Exp1]]\} \\ X[Exp1] := Exp2 \\ \{P\} \end{array}$$

Example 6 *We calculate the weakest precondition of the label $\{X[Z] = Y\}$ with regard to the program $X[Y] := 3$.*

$$\begin{array}{l} \{[Y = Z \wedge \#|_{\mathcal{O}w(X[Z])}?3 : X[Z]] = Y\} \\ X[Y] := 3 \\ \{X[Z] = Y\} \end{array}$$

Remark. If $Exp1$ or $Exp2$ contains parallel variables, we have to evaluate these expressions with regard to an active index u.

$$\begin{array}{c} \{P[\#|_u \wedge u = \mathcal{O}w(X[Exp1|_u])?Exp2|_u : X[Exp1|_u]/X[Exp1|_u]]\} \\ X[Exp1] := Exp2 \\ \{P\} \end{array}$$

The sequence $S;T$. We want to calculate the weakest precondition of a label P for the program $S;T$.

$$\frac{\{R\}S\{Q\},\ \{Q\}T\{P\}}{\{R\}\ S;T\ \{P\}}$$

The conditional structure $\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}$. We build the weakest precondition of a label P for this conditional structure. First, we suppose that the label Q does not depend on the activity $\#$. In this case, the execution of the conditional structure corresponds to the execution of S with a new activity $\# \wedge B$ [5].

$$\frac{\{P\}S\{Q\},\ \# \notin Var(Q)}{\{P[\# \wedge B/\#]\}\ \mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}\ \{Q\}}$$

The general case can be reduced to the previous case by renaming the activity $\#$ appearing in Q by a variable Tmp [6]. Assuming that the parallel variable Tmp is a new one, we apply the previous rule.

$$\frac{\{P\}S\{Q[Tmp/\#]\}}{\{P[\# \wedge B/\#]\}\ \textsf{where}\ \ B\ \ \textsf{do}\ \ S\ \ \textsf{end}\ \ \{Q[Tmp/\#]\}}$$

Since the activity is the same before and after the conditional structure, we substitute the variable Tmp in the label P by the activity $\#$. Note that the substitutions $[\# \wedge B/\#]$ and $[\#/Tmp]$ cannot commute.

Proposition 3 *Let Q be a label.*

$$\frac{\{P\}S\{Q[Tmp/\#]\}}{\{P[\# \wedge B/\#][\#/Tmp]\}\ \textsf{where}\ \ B\ \ \textsf{do}\ \ S\ \ \textsf{end}\ \ \{Q\}}$$

Proof 4 *The denotational semantics for the structure* where B do S end *is the same for the languages $\mathcal{L}$ and $\mathcal{L}'$ introduced in [8]. Considering the theorem 1, The demonstration corresponds to the proof of the proposition 7 in [8].*

The conditional structure where B do S elsewhere T end**.** The rule of the conditional structure where / elsewhere can be deduced from the previous one by transforming the program. We translate the program in a equivalent sequence of three conditional structures. The first one introduces temporary variables. In the second one, we modify the program S in order to reference and assign the temporary variables. The third one updates the original variables thanks to the temporary variables.

Example 7 *Thanks to the previous method, we transform the left hand side program to the right hand side one. We denote by $TmpX$ the temporary variable corresponding to the variable X. Note that we must have : $\mathcal{O}w(X) = \mathcal{O}w(TmpX)$*

where $This = 1$ do
  $X := X + 1$
elsewhere
  $Y := X$
end
	where $This = 1$ do
	  $TmpX := X;$
	  $TmpX := TmpX + 1$
	end;
	where $This \neq 1$ do
	  $Y := X$
	end;
	where $This = 1$ do
	  $X := TmpX;$
	end

We can transform the label $\{X = Y\}$ taking place after the where/elsewhere *structure with regard to the initial state. We use the transformed program. The label goes back from its current position to the beginning of the program.*

$\{[\#|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = 1?X + 1 : X] \ = \ [\#|_{\mathcal{O}w(Y)} \wedge \mathcal{O}w(Y) \neq 1?X : Y]\}$
where $This = 1$ do
  $\{[Tmp|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = 1?[\#|_{\mathcal{O}w(X)}?X + 1 : TmpX] : X]$
    $= \ [Tmp|_{\mathcal{O}w(Y)} \wedge \mathcal{O}w(Y) \neq 1?X : Y]\}$
  $TmpX := X;$
  $\{[Tmp|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = 1?TmpX + [\#|_{\mathcal{O}w(X)}?1] : X]$
    $= \ [Tmp|_{\mathcal{O}w(Y)} \wedge \mathcal{O}w(Y) \neq 1?X : Y]\}$
  $TmpX := TmpX + 1$
  $\{[Tmp|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = 1?TmpX : X] \ = \ [Tmp|_{\mathcal{O}w(Y)} \wedge \mathcal{O}w(Y) \neq 1?X : Y]\}$
end;
$\{[\#|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = 1?TmpX : X] \ = \ [\#|_{\mathcal{O}w(Y)} \wedge \mathcal{O}w(Y) \neq 1?X : Y]\}$
where $This \neq 1$ do
  $\{[Tmp|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = 1?TmpX : X] = [\#|_{\mathcal{O}w(Y)}?X : Y]\}$
  $Y := X$
  $\{[Tmp|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = 1?TmpX : X] = Y\}$
end;
$\{[\#|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = 1?TmpX : X] = Y\}$
where $This = 1$ do
  $\{[\#|_{\mathcal{O}w(X)}?TmpX : X] = Y\}$
  $X := TmpX;$
  $\{X = Y\}$
end
$\{X = Y\}$

Label occurring in a conditional structure. We can define the weakest precondition corresponding to a label occurring in a branch where (resp. elsewhere) of a conditional structure. These rules restrict the activity in the label Q to the indices which evaluate the condition to true (resp. false). The proof of correctness relies on the substitution lemma [1] like for the proposition 2.

$$\{P[\# \wedge B/\#]\} \ \text{where} \ B \ \text{do} \ \{Q\}$$

$$\frac{\{P\} \ S \ \{Q\}}{\{P[\# \wedge B/\#]\} \ \text{where} \ B \ \text{do} \ S\{Q\}}$$

$$\{P[\# \wedge \neg B/\#]\} \ \text{where} \ B \ \text{do} \ S \ \text{elsewhere} \ \{Q\}$$

$$\frac{\{P\} \ T \ \{Q\}}{\{P[\# \wedge \neg B/\#]\} \ \text{where} \ B \ \text{do} \ S \ \text{elsewhere} \ T \ \{Q\}}$$

Remark. As for the denotational semantics (cf. remark page 11), we apply the same priority order on these rules.

Example 8 *Let turn to the previous example 7. We consider a new label $\{X = Y\}$ occurring in the second branch and we deduce the corresponding label with regard to the initial state.*

  $\{X = [\#|_{\mathcal{O}w(Y)} \wedge \mathcal{O}w(Y) \neq 1?X : Y]\}$
where $This = 1$ do
  $X := X + 1$
elsewhere
  $\{X = [\#|_{\mathcal{O}w(Y)}?X : Y]\}$
  $Y := X$
  $\{X = Y\}$
end

Example 9 *Let turn to the running example 1. We express each label with regard to the initial state. Each label E_i goes back from its current position to the beginning of the program. We obtain the following stamped program.*

$E_1,\ E_2:\ \{\#|_v \wedge u = v = Y\}$
$E_3:\ \{\#|_v \wedge (u = v \vee u = \mathcal{O}w(X))\}$
$E_4,\ E_5:\ \{\#|_v \wedge u = v = [\#|_{\mathcal{O}w(X)} \wedge \mathcal{O}w(X) = Y?p+1 : X]\}$
where $This = Y$ do
$E_1,\ E_2:\ \{\#|_v \wedge u = v\}$
$E_3:\ \{Tmp|_v \wedge (u = v \vee u = \mathcal{O}w(X))\}$
$E_4,\ E_5:\ \{Tmp|_v \wedge u = v = [\#|_{\mathcal{O}w(X)}?p+1 : X]\}$
$X := p + 1$
$E_2:\ \{\#|_v \wedge u = v\}$
$E_3:\ \{Tmp|_v \wedge (u = v \vee u = \mathcal{O}w(X))\}$
$E_4,\ E_5:\ \{Tmp|_v \wedge u = v = X\}$
end;
$E_3:\ \{\#|_v \wedge (u = v \vee u = \mathcal{O}w(X))\}$
$E_4,\ E_5:\ \{\#|_v \wedge u = v = X\}$
where $X = This$ do
$E_4,\ E_5:\ \{\#|_v \wedge u = v\}$
$Y := 1;$
$E_5:\ \{\#|_v \wedge u = v\}$
end

3.3.1 Computing dates

Once each label E_i is transformed with regard to the initial state, we can substitute it in the graph by its weakest precondition. We can compute a symbolic cost.

We label each node $Inst$ of the graph by a parallel date δ. Each date $\delta(u)$ evaluated from the initial state corresponds in the execution graph to the length of the longest path ending at the node $(Inst, u)$ in the execution graph.

Let suppose that $Inst$ has m direct predecessors with respective dates δ_1, ..., δ_m. The arrow between each predecessor i and the node $Inst$ is labeled by the condition C_i.

The execution time A for an instance of instruction $Inst$ is evaluated to 1 if $Inst$ is meaningful, i.e. it performs an evaluation or an assignment, 0 otherwise. If v is an active index, i.e. if there exists a valid dependence of sequence $(C_i(v, v))$, then the date $\delta(v)$ associated is the maximum on the predecessor dates, increased by the execution time A. Otherwise, The resulting date is the maximum of the $\delta_i(v)$.

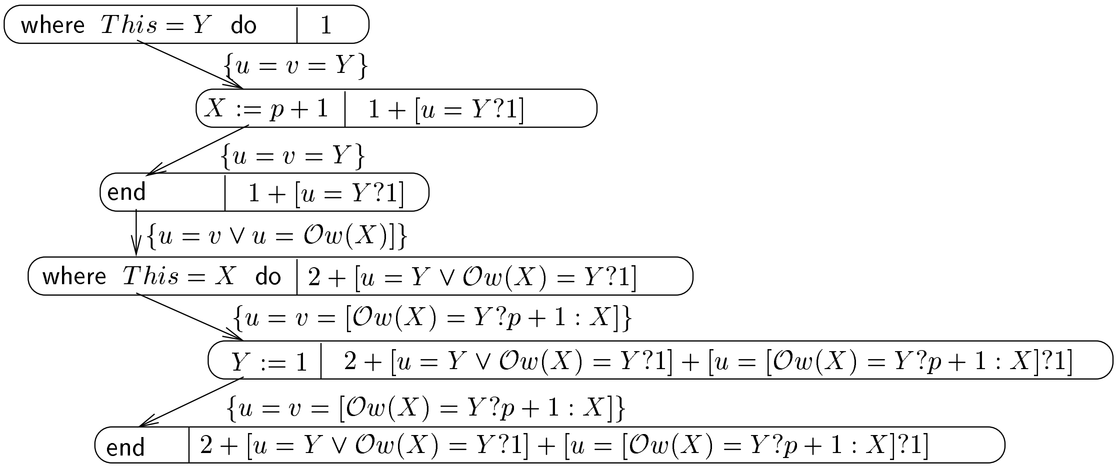
$$\delta(v) = \bigvee_{i\in[1..m]} C_i(v,v)? \max_{i\in[1..m],u\in[1..p]}\{[C_i(u,v)?\delta_i(u)]\} + A : \max_{i\in[1..m]}\{\delta_i(v)\}$$

We define the symbolic cost for a $\mathcal{SCP}$ program thanks to these dates.

Definition 3 (Symbolic cost.) *We denote by $Cost_s(S)$ the symbolic cost for a program S corresponding to the maximum on the final dates $\delta(u)$ for all the indices u.*

$$Cost_s(S) = \max_{u\in[1..p]}\{\delta(u)\}$$

Example 10 *Let turn to the running example 1. We obtain the following graph where each node is labeled by a date.*



We can deduce the following symbolic cost.

$$Cost_s(S) = \max_{u \in [0..p-1]} \{2 + [u = Y \vee \mathcal{O}w(X) = Y?1] + [u = [\mathcal{O}w(X) = Y?p + 1 : X]?1]\}.$$

The evaluation of the symbolic cost with regard to a given state corresponds to the cost of the corresponding execution.

Theorem 2 *Let S be a program and σ be a state. The evaluation of the symbolic cost of S with regard to the σ is equal to the cost of the corresponding execution.*

Proof 5 *For sake of clarity, we consider the symbolic dates on the expanded potential dependence graph: each node $(inst, v)$ is labeled by a date $\delta(v)$. The proof can be deduced from the construction of the symbolic dates.*

$$\delta(v) = \bigvee_{i \in [1..m]} C_i(v, v)? \max_{i \in [1..m], u \in [1..p]} \{[C_i(u, v)?\delta_i(u)]\} + A : \max_{i \in [1..m]} \{\delta_i(v)\}$$

In evaluation of a symbolic date, we identify the different steps for the construction of the execution graph.

First, we do not take into account the $(inst, v)$ satisfying one of the two following conditions:

- *Either, the execution time A of inst is equal to 0. This corresponds to instructions inst not meaningful (i.e. inst is only a syntactic marker).*
- *Either, the expression $\bigvee_{i \in [1..m]} C_i(v, v)$ is evaluated to false. There is no valid sequence dependence ending at this node. Therefore, the index v is not active with regard to inst in the corresponding execution.*

In these cases, the date of the node can be reduced to maximum of the dates of its predecessors($\max_{i \in [1..m]} \{\delta_i(v)\}$). This allows to preserve the maximum date for its successors. Therefore, we take into account all the dependences needed to execute other instructions, i.e. the set S of the arrows starting from this node.

Finally, we does not take into account the arrows from $(source, u)$ to $(target, v)$ labeled by $C(u, v)$ such that $C(u, v)$ is evaluated to false. By construction of the label, this arrow corresponds a flow dependence that does not occur in the execution (i.e. where $u \neq v$ and, where there is no variable of u accessed during the execution of $target$ by v).

Therefore, for a given node, the evaluation of the symbolic dates corresponds to the length of the longest path ending at this node.

Example 11 *Let turn to the previous example. We evaluate the symbolic cost with regard to the initial state.*

$$\begin{aligned}\sigma(Cost_s(S)) &= \max_{u\in[0..P-1]}\{2+[u=Y \vee \mathcal{O}w(X)=Y?1]+[u=P+1?1]\}\\ &= \max_{u\in[0..P-1]}\{2+[u=0 \vee \mathcal{O}w(X)=0?1]\}\\ &= 3\end{aligned}$$

This evaluation corresponds to the execution cost computed in the example 3.

4 Composing costs: the symbolic semantics

In the previous section, we introduced a model computing a symbolic cost for a program. To associate a cost to a loop by using the fix point method, we need to compose the costs. Each symbolic cost refers to the initial state of the corresponding program. Therefore, to compose two costs, we have to transform them with regard to a common state. This would require a weakest precondition calculus able to transform expression while it only transforms predicates.

We propose a symbolic semantics extending the weakest preconditions calculus to expressions. Let $\mathcal{E}$ be the set of expressions. We define the symbolic semantics $\|S\|$ of the program S as a function from $\mathcal{E}$ to $\mathcal{E}$. If the program execution does not end, we introduce a default value, denoted by $V_\perp$, for each type of variable. A complete presentation of this semantics can be found in [13].

Definition 4 *Let S be an $\mathcal{SCP}$ program and Exp be an expression. We call symbolic semantics of S, denoted by $\|S\|$, a function from the expression set into itself. It satisfies for each state σ*

$$\sigma(\|S\|(Exp)) = \begin{cases} [\![S]\!](\sigma)(Exp) \text{ if the execution of } S \text{ ends with regard to } \sigma,\\ V_\perp \text{ (default value) otherwise.}\end{cases}$$

Remark. If the expression Exp is a predicate, and if we choose $V_\perp = True$, then $\|S\|(Exp)$ is the weakest liberal precondition $wlp(S, Exp)$ [13].

Example 12 *Let $X := X+1$ be a $\mathcal{SCP}$ program and $X+7$ be an expression. The corresponding symbolic semantics has to substitute by $X+1$ all the occurrences of X. We obtain*

$$\|X := X+1\|(X+7) = (X+7)[X/X+1] = (X+8)$$

To construct the symbolic semantics of a loop, we use the corresponding denotational semantics as a least fixed point. We first define an operator, denoted by $\bigcirc$, to compose symbolic semantics. It transforms a symbolic semantics sb into an other one.

$$\begin{aligned} \bigcirc_{k=1}^{1} sb(Exp) &= sb(Exp)\\ \text{For } m>1,\quad \bigcirc_{k=1}^{m} sb(Exp) &= \bigcirc_{k=1}^{m-1} sb(sb(Exp))\end{aligned}$$

For each loop $\mathsf{loop}\ B\ \mathsf{do}\ S\ \mathsf{end}$, we define upper level order function, denoted by $\bigcirc_{k=1}^{NbIter} \|\ \mathsf{where}\ B\ \mathsf{do}\ S\|$. Intuitively, such function corresponds to the composition $NbIter$ times of the semantics of an iteration. By definition of the least fixed point [14], for each initial state σ such as the loop ends, there exists a least integer k such that $Fix(F)(\sigma) = F^k(\sigma)$ (k depends on σ). Therefore, we define $NbIter$ as a new variable evaluated to k with regard to σ.

Example 13 *Let* $\mathsf{loopwhere}\ (X<5)\ \mathsf{do}\ X := X+1\ \mathsf{end}$ *be a $\mathcal{SCP}$ program and $X+7$ be an expression. We first compute the symbolic semantics to the part* $\mathsf{where}\ (X<5)\ \mathsf{do}\ X := X+1$.

$$\|\ \mathsf{where}\ (X<5)\ \mathsf{do}\ X := X+1\|(Exp) = Exp\ [\#/\# \wedge (X<5)]\ [X/[\#|_{\mathcal{O}w(X)}?X+1:X]]$$

We imposes that the owner of X is initially active. Therefore, the loop ends for all initial state σ such as $\sigma(X) + \sigma(NbIter) * 1 = 5$. We substitute $NbIter$ by the expression $5 - X$.

$$\begin{aligned}
&\| \textsf{loopwhere } (X < 5) \textsf{ do } X := X + 1 \textsf{ end } \|(Exp) \\
&= (\bigcirc_{k=1}^{5-X} \| \textsf{where } (X < 5) \textsf{ do } X := X + 1 \|)(Exp) \\
&= (Exp)[X/[(X < 5)?X + 1 * (5 - X) : X]]
\end{aligned}$$

If we apply this function to the expression $X + 7$, we obtain

$$\| \textsf{loopwhere } (X < 5) \textsf{ do } X := X + 1 \textsf{ end } \|(X + 7) = [(X < 5)?X + 1 * (5 - X) : X] + 7$$

Thanks to the symbolic semantics, we are able to transform a symbolic cost with regard to a new initial state. We now have to define the rules to compose two costs. During the composition, in order to take into account the asynchronism, we have to preserve the final gap between the dates of indices. Rather than considering the maximum of the final date of each index ($max_{u \in [1..P]}\{\delta'(u)\}$), we define a complexity function, denoted by $Comp$ computing the symbolic date (δ') of the final node in the graph. In order to compose it, the $Comp$ function must use an initial date for the first node of the graph.

$$Comp(S)(\delta) = \delta'$$

From this complexity, we deduce the cost for the program S by performing the maximum of the final date.

$$Cost(S) = max_{u \in [0..p-1]}\{Comp(S)(0)(u)\}$$

We can now define a composition rule for the sequence $S;T$. First, we transform the final date of $Comp(T)$ with regard to the initial state of S. Then, we use the final date of $Comp(S)$ as the initial date of $Comp(T)$.

$$Comp(S;T)(\delta) = \|S\|(Comp(T))[Comp(S)(\delta)]$$

For sake of simplicity, we introduce in the potential graph a first node called start stamped by the initial date $\delta(u)$.

Example 14 *Let turn to the program $X := X + 1; X := Y[X]$. We first construct two labeled graphs for the two instructions. Then, we compute the dates.*

start	$\delta_1(u)$
	$\downarrow \{\#_v \wedge u = v\}$
$X := X + 1$	$\delta_1(u) + [\#\vert_u?1]$

start	$\delta_2(u)$
	$\downarrow \{\#\vert_v \wedge (u = v \vee (v = \mathcal{O}w(X) \wedge u = \mathcal{O}w(Y[X])))\}$
$X := Y[X]$	$[\#\vert_u?max\{\delta_2(u); \delta(\mathcal{O}w(Y[X]))\} + 1 : \delta_2(u)$

We obtain:

$$\begin{aligned}
&Comp(X := X + 1)(\delta_1) = \delta_1' \\
&\quad where\ \delta_1'(u) \ = \ \delta_1(u) + [\#|_u?1] \\
&Comp(X := Y[X])(\delta_2) = \delta_2' \\
&\quad where\ \delta_2'(u) \ = \ [\#|_u?max\{\delta_2(u), \delta_2(\mathcal{O}w(Y[X])\} + 1 : \delta_2(u)]
\end{aligned}$$

We first express the final date for $X = Y[X]$ with regard to the initial state of $X = X + 1$.

$$\begin{aligned}
&\|X := X + 1\|(Comp(X := Y[X]))(\delta_2) = \delta_2' \\
&\quad where\ \delta_2'(u) \ = \ [\#|_u?max\{\delta_2(u), \delta_2(\mathcal{O}w(Y[X + 1])\} + 1 : \delta_2(u)]
\end{aligned}$$

Then, we can compose the two complexity functions.

$$\begin{aligned}
&Comp(X := X + 1; X := Y[X])(\delta) = \delta' \\
&\quad where\ \delta'(u) \ = \ [\#|_u?max\{\delta(u) + 1, \delta_2(\mathcal{O}w(Y[X + 1]) + [\#|_{\mathcal{O}w(Y[X])}?1]\} + 1 : \delta_2(u)]
\end{aligned}$$

If we consider that the initial date is 0 *and all the indices initially active, we can compute the cost for the program as the maximum of the final dates of each index:* $Cost(X = X + 1; X = Y[X]) = 2$.

We now construct a complexity function for the loops. Before composing iterations, the final symbolic date of each iteration has to be expressed with regard to the initial state. We define an operator to compose complexities, denoted by $\bigotimes$. Note that we impose $V_\perp = \infty$ for the symbolic dates.

$$\bigotimes_{k=1}^{1} Comp(\mathsf{where}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end})(\delta) \ = \ Comp(\mathsf{where}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end})(\delta)$$

for $m > 1$,

$$\begin{array}{l} \bigotimes_{k=1}^{m} Comp(\mathsf{where}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end})(\delta) \ = \\ \quad [\bigcirc_{k=1}^{m-1} \| \mathsf{where}\ \ B\ \ \mathsf{do}\ \ S \| Comp(\mathsf{where}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end})](\bigotimes_{k=1}^{m-1} Comp(\mathsf{where}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end})(\delta)) \end{array}$$

Thanks to this operator, we can define the complexity function for the loop as a composition $NbIter$ times of the complexity of the conditional structure.

Example 15 *Let turn to the example 13. To compute the complexity, we first express the complexity of an iteration.*

$$Comp(\mathsf{where}\ \ X < 5\ \ \mathsf{do}\ \ X := X + 1\ \mathsf{end})](\delta) \ = \ \delta'$$

where

$$\delta'(u) = \delta(u) + [\#|_u?1 + [(X < 5)?1]]$$

Next, we transform the complexity of an iteration l $(0 < l < 5 - X)$ *with regard to the initial state. Since we consider.*

$$\begin{array}{rcl} F_l(\delta) & = & \bigcirc_{j=1}^{l-1} \| \mathsf{where}\ \ X < 5\ \ \mathsf{do}\ \ X := X + 1 \| \\ & & Comp(\mathsf{where}\ \ X < 5\ \ \mathsf{do}\ \ X := X + 1\ \mathsf{end})](\delta) \ = \ \delta'' \end{array}$$

where

$$\begin{array}{rl} \delta''(u) & = \delta(u) + ([\#|_u?1 + [(X < 5)?1]])\ [\#/[l = 1?\# : \# \wedge (X < 5)]]\ [X/[(X < 5)?X + l : X]] \\ & = \delta(u) + [[l = 1?\#_u : \#|_u \wedge (X < 5)]?1 + [(X < 5)?1]] \end{array}$$

We compose the different complexities expressed with regard to the initial state.

$$\begin{array}{l} Comp(\mathsf{loopwhere}\ \ X < 5\ \ \mathsf{do}\ \ X := X + 1\ \mathsf{end})(\delta) \\ \quad = F_{NbIter} \circ \ldots \circ F_1(\delta) \ = \delta' \end{array}$$

where

$$\begin{array}{rcl} \delta'(u) & = & \delta(u) + 1 + [(X < 5)?2 * NbIter - 1] \\ & = & \delta(u) + 1 + [(X < 5)?2 * (5 - X) - 1] \end{array}$$

We can deduce the following cost

$$Cost(\mathsf{loopwhere}\ \ X < 5\ \ \mathsf{do}\ \ X := X + 1\ \mathsf{end}) \ = \ 1 + [(X < 5)?2 * (5 - X) - 1]$$

Remark. Thanks to these composition rules, we can define a one pass computation of labels and dates based on a induction on the program structure [13].

5 Conclusion

We have presented a cost model for the $\mathcal{SCP}$ programs. We show it is possible to compute a symbolic cost taking into account static independences defined in the program and dynamic independences due to different initial states.

In order to identify static independences, we consider the complementary set: the potential dependences. We construct a potential dependence graph. In the $\mathcal{SCP}$ model, the structuration of dependences ensures that the potential dependence graph is a direct acyclic one. Therefore, the cost of a program can be classically defined as the length of the longest path in the corresponding graph.

In order to to consider dynamic independences, we label each arrow of the graph by a predicate. Each label symbolizes if the corresponding dependence occurs with regard to a given state. Thanks to these labels, we can compute a symbolic cost which can be evaluated regarding to an initial state to obtain the cost of the corresponding execution. We perform it construction in two passes. First, we use the weakest preconditions calculus to express the labels with regard to the initial state. Then, we use the transformed labels to compute the symbolic cost.

We show that it is possible to compose complexities We define a symbolic semantics extending the weakest liberal precondition calculus to expressions. Thanks to this semantics, we can perform the cost calculus by induction on the the program structure.

These properties makes the $\mathcal{SCP}$ model a good study model for automatic code distribution [9, 11, 12]. We are currently working on the extension of the approach to general loops. The adaptation of this approach to other models structuring the dependences (like Athapascan) is under progress. In order to take into account different types of computers, we plan to introduce architecture parameters in the symbolic cost.

References

[1] BK.R. Apt and E-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[2] G. Cavalheiro, F. Galilee, and J.-L. Roch. Athapascan-1: Parallel programming with asynchronous tasks. In *Proc. Yale Multithreaded Programming Workshop*, 1998.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and V. E. Thorsten. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, May 1993.

[4] E. W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *CACM*, 18(8):453–457, August 1975.

[5] J. Gabarró and R. Gavaldà. An approach to Correctness of Data-Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22(1):185–201, 1994.

[6] Y. Le Guyadec and B. Virot. Sequential-like Proofs of Data-Parallel Programs. *Parallel Processing Letters*, 6(3):415–426, 1996.

[7] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. Horloges structurelles pour la désynchronisation de programmes data-parallèles. In *Actes de RenPar'8*, pages 77–80. Université de Bordeaux, France, may 1996.

[8] Yann Le Guyadec and Bernard Virot. An Axiomatic Semantics of Conditioning Constructs and Non Local Control Transfer in Data-Parallel Languages. Technical Report RR94-15, LIFO, Orléans, France, 11 1994.

[9] E. Melin. *Traitement de l'irrégularité dans la parallélisation de code séquentiel par distribution des données*. PhD thesis, Université d'Orléans, Orléans, France, 1998.

[10] E. Melin, B. Raffin, X. Rebeuf, and B. Virot. A Cost Model For Asynchronous and Structured Message Passing. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, volume 1685 of *LNCS*, pages 552–556. Springer-Verlag, 1999.

[11] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Simple Synchronization and Communication Multi-threaded Library for Automatic Distribution of Irregular Sequential Code. In *Third International Conference on Massively Parallel Computing Systems - MPCS'98*, pages 482–489, Colorado Springs, USA, April 1998.

[12] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Structured Synchronization and Communication Model Fitting Irregular Data Accesses. *Journal of Parallel and Distributed Computing*, 50(1):3–27, 1998.

[13] X. Rebeuf. *Un modèle de coût symbolique pour les programmes parallèles asynchrones à dépendances structurées*. PhD thesis, Université d'Orléans, Orléans, France, 2000.

[14] David A. Schmidt. *DENOTATIONAL SEMANTICS: A Methodology for Language Development*. Wm. C. Brown, 1986.

[15] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.